UMF – UI Manipulation Framework

# Inhaltsverzeichnis

[1 Inhaltsverzeichnis 1](#_Toc268852463)

[2 Was ist UMF? 2](#_Toc268852464)

[3 Kontextmanagement 2](#_Toc268852465)

[3.1 Konfiguration des ContextManagers 2](#_Toc268852466)

[3.2 Anwendung eines Kontexts auf ein Objekt 2](#_Toc268852467)

[3.3 Die Struktur der Konfigurationsdatei 3](#_Toc268852468)

[3.3.1 Grundlagen 3](#_Toc268852469)

[3.3.2 Das Action-Element 4](#_Toc268852470)

[3.3.3 Das ModifiedProperty-Element 4](#_Toc268852471)

[3.3.4 Das InvokedMethod-Element 4](#_Toc268852472)

[3.3.5 Ausführungsreihenfolge 5](#_Toc268852473)

[3.4 Benutzerdefinierte Aktionen 5](#_Toc268852474)

[3.4.1 Implementierung benutzerdefinierter Aktionen 5](#_Toc268852475)

[4 Factories 7](#_Toc268852476)

[4.1 Das ConnectionAttribute 7](#_Toc268852477)

[4.2 ValueConverter 8](#_Toc268852478)

[4.3 ConnectionFlags 8](#_Toc268852479)

[4.4 DefaultValueProvider 8](#_Toc268852480)

[4.5 Beispiel: 9](#_Toc268852481)

[5 Layoutmanager 11](#_Toc268852482)

[5.1 Anwendung des Layoutmangers 11](#_Toc268852483)

[5.2 Integration ist Kontextmanagement 12](#_Toc268852484)

[5.3 Der Layoutvorgang 12](#_Toc268852485)

[5.4 Steuerung des Layoutvorgangs über Panels 13](#_Toc268852486)

[5.5 Weitere Steuerungsmöglichkeiten 13](#_Toc268852487)

[5.6 Pitfalls 14](#_Toc268852488)

# Was ist UMF?

Das UiManipulationFramework stellt primär Funktionen zur kontextabhängigen Manipulation von Benutzeroberflächen zur Verfügung. Über eine XML-Konfigurationdatei können verschiedene Kontexte definiert werden, die Konfigurationregeln für Klassen enthalten. Hiermit ist es möglich die Sichtbarkeit, die Sprache und den Status von GUI-Elementen zu steuern.

In Verbindung mit den Layoutfunktionen kann die Anzeige der GUI-Elemente reorganisiert werden, so dass z.B. Lehrräume automatisch gepack, oder überlappende Elemente neu angeordnet werden. Das Layouting wurde „nicht invasiv“ gestaltet, es ist nicht notwendig bereits bestehende GUIs auf das Framework anzupassen, es müssen lediglich einige Zeilen Code hinzugefügt werden.

Im Falle von Eingabemasken, stellt sich nun ein Problem. Oft ist das Ergebnis einer Eingabe ein Objekt, das aus diesen Daten erzeugt wird. Kontextabhängig kann sich die Verfügbarkeit der anfallenden Daten unterscheiden, folglich muss überprüfet werden, welche Daten zur konfiguration eines Objektes zur Verfügung stehen. Dieses Problem wird durch kontextabhängige generische Factories gelöst, die in Abhängigkeit des gewählten Kontexts aus der Eingabe ein Objekt erstellen können. Die Richtung lässt sich auch umkehren, um mittels den Daten eines Objektes die GUI auszufüllen.

Die Möglichkeiten der Kontext- und Factoryfunktionen lassen sich im Endeffekt nicht nur auf GUI-Objekte sondern auf jedes beliebige Objekt anwenden.

# Kontextmanagement

## Konfiguration des ContextManagers

Das Herzstück des Kontextmanagement ist das Interface IContext und dessen Implementiertung XMLContext, die im ContextManager verwaltet werden. Zur Registrierung der Kontexte aus einer XML-Datei dient der XMLConfigurator, der die statische Methode Configure(String path) zur Verfügung stellt.

UBT.AI4.Bio.DivMobi.umf.context.XMLConfigurator.Configure(@"\Storage Card\conf.xml");

Der Aufruf bewirkt, dass alle in der Datei conf.xml gemachten Kontextdefinitionen im ContextManager registriert werden. Es ist möglich mit mehreren sukzessiven Aufrufen von Configure verschiedene Konfigrationsdateien zu laden, dabei werden jedoch gegebenenfalls gleichnamige Kontexte überschrieben.

## Anwendung eines Kontexts auf ein Objekt

Der KontextManager kann nach der Konfiguration verwendet werden, um registrierte Kontexte ausfindig zu machen. Der KontextManager ist als Singleton implementiert, somit kann an jeder Stelle des Programms auf ihn zugegriffen werden. Die Kontextsuche erfolgt über die Methode GetContext(String contextId).

IContext context = UBT.AI4.Bio.DivMobi.umf.context.ContextManager.Instance.GetContext("id");

Der Methodenaufruf liefert ein Objekt vom Typ IContext zurück, das die Methode Configure(Object o) anbietet. Der Aufruf der Methode mit einem Objekt als Parameter bewirkt, sofern der Typ des Objekts im Kontext registriert wurde, die Anwendung des Kontexts auf das Objekt.

MyObject o = new MyObject ();

context.Configure(o);

Der Kontext führt nun automatisch alle für den Objekttyp definierten Aktionen aus. Falls der Objekttyp nicht registriert ist bewirkt der Aufruf nichts.

## Die Struktur der Konfigurationsdatei

### Grundlagen

Die Konfiguration erfolgt über eine oder mehrere XML-Dateien, die den im Folgenden beschriebenen Aufbau haben müssen. Das Wurzelelement der Datei ist ContextRoot, in dem mehrere Kontexte über das Element Context definiert werden können.

<ContextRoot>

<Context ContextId="id">

…

</Context>

</ContextRoot>

Es gilt zu beachten, dass das Attribut ContextId des Elements Context erforderlich ist.

Anmerkung: Das Fehlen unerlässlicher Attribute wird momentan nicht beim Parsen erkannt, da das verwendete XML-Serialisierungs-Framework keine Überprüfung durchführt! Nach dem Laden der Konfiguration wird jedoch deren Gültigkeit überprüft und im Problemfall eine ContextConfigurationException ausgelöst.

Das Element Context kann wiederum mehrere ClassConfiguration-Elemente enthalten.

<Context ContextId="id">

<ClassConfiguration ClassName=”Test.MyObejct, Test, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">

…

</ClassConfiguration>

</Context>

Hier ist das Attribut ClassName obligatorisch, der vergebene Name muss dem „assembly-qualified name“ der zu konfigurierenden Klasse entsprechen. Die Angabe eines ungültige Namens führt zu einer ContextConfigurationException. Innerhalb der ClassConfiguration werden die zu modifizierenden Felder mittels des Elements FieldDescriptor angegeben, das auch mehrfach vorkommen darf.

<ClassConfiguration ClassName="MyObject">

<FieldDescriptor FieldName="\_myObjectsField">

…

</FieldDescriptor>

</ClassConfiguration>

Das Attribut FieldName muss der Name eines Feldes innerhalb der zuvor referenzierten Klasse sein. Wenn das entsprechende Feld in der Klasse nicht existiert wird während dem Lesen der Konfiguration eine ContextConfigurationException ausgelöst.

Das Framework unterstützt drei Möglichkeiten Felder zu manipulieren:

Ausführen einer Aktion (Action- Element)

Modifikation einer Property des Feldes (ModifiedProperty- Element)

Aufruf einer Methode des Feldes (InvokedMethod- Element)

Anmerkung: die Verwendung der ModifiedProperty- und InvokedMethod-Elemente müssen durch Angabe einer Policy explizit erlaubt werden. Empfohlen wird jedoch die Verwendung von Actions, da diese explizit registriert werden müssen und somit sicherer sind als die beiden Varianten zuvor, außerdem entfällt hier die Notwendigkeit einer Policydefinition.

Alle der vier genannten Elemente können innnerhalb des FieldDescriptor-Elements in beliebiger Reihenfolge und Anzahl verwendet werden.

### ***Das*** *Action****-Element***

Das Action-Element definiert eine benutzerdefinierte Aktion, die während der Konfiguration eines Objekts über einen Kontext aufgerufen wird. Die Ausführung einer bestimmten Aktion erfolgt genau dann wenn während des Konfigurationsdurchlaufs eines Kontexts, das im FieldDescriptor beschriebene Feld gefunden wird und für dieses Feld eine Aktion definiert wurde.

<FieldDescriptor FieldName="\_myObjectsField">

<Action ActionId="UserdefinedAction" Parameter="true" />

</FieldDescriptor>

Das Attribut ActionId muss die ID einer registrierten Aktion sein, falls die ID nicht im System bekannt ist, wird das System zur Laufzeit eine ContextRuntimeException auslösen. Das Parameter –Attribut ist optional. Je nach Implementierung der Aktion (siehe unten) kann dies jedoch zu einem Laufzeitfehler führen der von der Implementierung behandelt werden muss.

### Das *ModifiedProperty*-Element

Die Definition eines ModifiedProperty-Elements, ändert in einem Konfigurationsdurchlauf den Wert der referenzierten durch das PropertyName-Attribut referenzierten Property. Falls die Property nicht für das im FieldDescriptor-Element angegebene Feld vorhanden ist, wird beim Laden der Konfiguration eine ContextConfigurationException ausgelöst. Das Attribut PropertyName ist zwingend erforderlich, nicht jedoch das Attribut Value. Wird das Attribut nicht angegeben, so wird der Property der Wert null zugewiesen.

<FieldDescriptor FieldName="\_myObjectsField">

<ModifiedProperty PropertyName=" MyObjectsFieldProperty" Value="true" />

</FieldDescriptor>

Die Umwandlung des Wertes erfolgt für primitive Typen und Strings von Haus aus. Der Mechanismus bestimmt zuerst den Typ des Objekts, das der Property übergeben wird. Dann wird entsprechend des bestimmten Typs versucht der dem Value-Attribut zugewiesene String zu parsen. Falls der String sich nicht parsen lässt Wird eine ContextRuntimeException ausgelöst.

Die Umwandlung von Strings in beliebige Typen ist auch möglich, indem man einen entsprechenden Parser implementiert und registriert (siehe unten).

### Das *InvokedMethod*-Element

Die Definition dieses Feldes bewirkt den Aufruf der durch das MethodName-Attribut bestimmten Methode. Falls die Methode nicht existiert wird eine ContextConfigurationException beim laden der der Konfiguration ausgelöst. Momentan werden nur parameterlose Methoden unterstützt, die Konzeption und Erweiterung auf parametrisierte Methode sollte jedoch kein Problem darstellen.

<FieldDescriptor FieldName="\_myObjectsField">

<InvokedMethod MethodName="MyObjectFieldMethod" />

</FieldDescriptor>

### Ausführungsreihenfolge

Da es mehrere Möglichkeiten zur Manipulation eines Feldes gibt, muss deren Ausführungsreihenfolge bekannt sein. Momentan ist die Ausführungsreihenfolge fest vorgeben. Zuerst werden alle Actions, dann die Methodenaufrufe und abschließend die Propertymodifikationen ausgeführt.

## Benutzerdefinierte Aktionen

Aktionen sind meist relative kleine Codesegmente die währende eines Konfigurationsdurchlaufs eines Kontexts ausgeführt werden können und somit die kontextgestützte Konfiguration eines Objekts ermöglichen.

### Implementierung benutzerdefinierter Aktionen

Eine Aktion wird entweder durch die Implementierung des Interfaces IAction oder durch ableiten von AbstractAction definiert. Zweite Variante hat den Vorteil, dass sie bequemer ist und sollte in den meisten Fällen die gewählte Option sein.

#### Implementierung des Interfaces *IAction*

Klassen, die IAction implementieren, müssen zwei Methoden implementieren.

String ActionId { get; }

void Perform( Object fieldOwner,

FieldInfo fieldAccess,

Object fieldValue,

String parameter);

Erste Methode muss einen String zurückgeben, dieser muss eindeutig sein, darf also nicht mit anderen (registrierten) ActionIds überein stimmen. Nur wenn diese Regel eingehalten wird, kann eine eindeutige Zuordnung einer, in der Konfiguration referenzierten, Aktion gewährleistet werden.

Die Methode Perform ist etwas umfangreicher, ihre Implementierung legt das Verhalten der Aktion fest. Der Aufruf von Perform erfolg immer dann, wenn folgende Kriterien erfüllt sind:

Der Benutzer konfiguriert mit einem Kontext ein Objekt

Der Typ des Objekts hat eine ClassConfiguration in diesem Kontext

Das System erreicht bei der Konfiguration das Feld, für das die Aktion definiert wurde

Mit diesem Wissen lassen sich nun die Parameter der Methode Perform leicht erklären.

fieldOwner: das Objekt, das der Benutzer konfiguriert

fieldInfo: das FieldInfo, das referenzierte Feld erhält

fieldValue: der aktuelle Wert des Feldes

parameter: der Parameter-String der in der Konfiguration für diese Aktion angegeben wurde

Als Konsequenz daraus (in einem nicht nebenläufigen System) hat die Variable b im folgenden Codebeispiel immer den Wert true.

void Perform( Object fieldOwner,

FieldInfo fieldAccess,

Object fieldValue,

String parameter)

{

bool b = fieldAccess.GetValue(fieldOwner) == fieldValue;

}

Für das Zerlegen und Umwandel des Strings in die für die Implementierung der Methode notwendigen Variablen ist der Anwender des Frameworks selbst verantwortlich.

Theoretisch kann eine Aktion innerhalb vieler verschiedener ClassConfiguration-Elemente verwendet werden, entsprechend kann sich aber auch der Typ der Parameter fieldOwner und fieldValue ändern. Dieses Verhalten einer Aktion kann durchaus gewollt sein, allerdings muss der Anwender selber für eine angemessene Behandlung für unterschiedliche Typen sorgen.

#### Ableiten von *AbstractAction*

Diese Möglichkeit bietet mehrere Vorteile gegenüber der oben genannten Variante. Zum einen entfällt der oben genannte Typüberprüfung, zum anderen muss das Parsen des Parameterstrings (zumindest für primitive Datentypen) nicht mehr per Hand erfolgen. Das Umwandeln Strings in andere Datentypen ist aber auch hier möglich (siehe dazu 3.5).

Die abstrakte Klasse AbstractAction hat drei generische Typparameter: O, E, P

protected abstract class AbstractAction<O, E, P> : IAction

O ist der Typ für den die Aktion ausgelöst wird also der Typ, der in der ClassConfiguration referenziert wird.

E ist der Datentyp des Feldes, das im FieldDescriptor referenziert wird.

P ist der Typ, in den der Parameter der Action umgewandelt werden soll.

Um von AbstractAction abzuleiten, muss die abstrakte Methode PerformImpl implementiert werden zusätzlich wird dem Konstruktor, die ActionId in Form eines String übergeben. Die Signatur der abstrakten Methode sieht folgendermaßen aus:

protected abstract void PerformImpl( O fieldOwner,

FieldInfo fieldAccess,

E fieldValue,

P para);

Der Vorteil liegt auf der Hand, es nicht notwendig selber Typecasts oder Typüberprüfungen durchzuführen. Wird eine Aktion von einer inkompatiblen Klasse (innerhalb des ClassConfiguration-Elements) und / oder von einem falschen Feld (im FieldDescriptor-Element) referenziert führt das beim Aufruf der Aktion unweigerlich zu einer InvalidCastException.

Hinweis: Falls die, in beschriebene, Implementierung eines StringParsers für die Typumwandlung nicht erwünscht ist und die Umwandlung des Parameters „von Hand“ in der Implementierung von PerformImpl stattfinden soll, kann für P der Datentyp String eingesetzt werden. Dies führt dazu, dass der String unbearbeitet durchgereicht wird.

#### Registrierung der Aktionen

Damit eine Aktion referenziert werden kann muss sie dem System bekannt sein, zu diesem Zweck gibt es die ActionRegistry, sie dient dazu Benutzerdefinierte Aktionen zu registrieren und auch wieder zu entfernen. Die ActionRegistry ist als Singleton implementiert, die Referenz auf das Singletonobjekt erhält man über die Property Instance.

Die ActionRegistry stellt zwei Methoden zur Verfügung Register(IAction action) und Unregister(IAction action). Mit ersteren ist die Registrierung einer Aktion mit der zweiten die deregistrierung einer Aktion möglich.

Anmerkung: falls nicht registrierte Aktionen aufgerufen werden, wird eine ContextRuntimeException ausgelöst!

public class MyAction : AbstractAction<Object, Object, Int32>

{

…

}

…

public void AMethod()

{

IAction a = new MyAction();

ActionRegistry.Instance.RegisterAction(a);

//Die Aktion kann jetzt referenziert werden

ActionRegistry.Instance.UnregisterAction(a);

//Die Aktion kann nicht mehr referenziert werden

}

# Factories

Das An- bzw. Abschalten von GUI-Elementen bringt gegebenenfalls das Problem mit sich, dass manche Benutzerdaten für das initialisieren von Objekten nicht mehr zur Verfügung stehen. Auf der anderen Seite müssen GUI-Elemente, die deaktiviert sind nicht mit vorhandenen Daten initialisiert werden.

Zur Lösung dieses Problem bietet UMF das Konzept der *Initializer* an, die automatisch diese Aufgabe übernehmen. Es existieren zwei zentrale Klassen, diese Kopiervorgänge übernehmen, zum einen den *ControlConfigurator*, der Daten von einem Objekt zur GUI kopiert, und zum anderen die *DataItemFactory*, die mit Benutzerdaten ein Objekt erzeugt. Der Kopiervorgang wird über das *ConntectionAttribute* gesteuert, tatsächlich werden nur GUI-Elemente, die mit diesem Attribute versehen sind, bei dem Kopiervorgang einbezogen.

## Das ConnectionAttribute

Mit Hilfe des ConnectionAttributes werden Properties von GUI-Elementen mit Properties von Objekten verbunden. Der Konstruktor des ConnectionAttributes erwartet hat vier obligatorische Parameter und zwei optionale Parameter.

ConnectionAttribute(Type connectedType, String controlProperty, String dataItemProperty, Type valueConverter);

* connectedType gibt an mit welchen Objekt das GUI-Element verbunden werden soll
* controlProperty gibt an welche Property des GUI-Elements beim kopieren verwendet werden soll
* dataItemProperty gibt an welche Property des DataItems beim kopieren verwendet werden soll
* valueConverter gibt an welcher ValueConverter verwendet werden soll, um Daten umzuwandeln

Folgende Parameter sind Optional:

* ConnectionFlag: steuert das kopieren von Daten wenn ein GUI-Element unsichtbar oder deaktiviert ist. In der Grundeinstellungen werden Daten von unsichtbaren und / oder deaktivierten GUI-Elementen nicht kopiert
* DefaultValueProvider: im Fall, dass Daten nicht kopiert werden, kann mit diesem Parameter ein Typ angeben werden, der ein Defaultwert zur Verfügung stellt

## ValueConverter

Bei den meisten Benutzereingaben handelt es sich um Strings. Oft ist es aber so, dass diese in einen anderen Datentyp umgewandelt werden müssen. Diese Umwandlung muss beim Kopieren der Daten passieren. Aus diesem Grund wird in UMF das Interface *IValueConverter* zur Verfügung gestellt. Der Typ, der dem Konstruktor des *ConnectionAttribute*s als valueConverter übergeben wird muss sich von diesem Interface ableiten. Das Interface erfordert die Implementierung von zwei Methoden:

* public Object ConvertForDataItem(Object val)
* public Object ConvertForControl(Object val)

Erstere übernimmt die Konvertierung von Benutzereingabedaten in ein für das DataItem kompatibles Format. Die zweite übernimmt die umgekehrte Richtung.

**Achtung: Der Konstruktor des von IValueConverter erbenden Typs muss öffentlich und parameterlos sein.**

## ConnectionFlags

Mit den ConnectionFlags kann der Kopiervorgang gesteuert werden. Es gibt momentan zwei Flags, die gesetzt werden können:

* ConnectionFlags.CopyOnInvisible: kopiert Daten von unsichtbaren GUI-Elementen, bzw. initialisiert unsichtbare GUI-Elemente
* ConnectionFlags.CopyOnDisabled: kopiert Daten von deaktivierten GUI-Elementen, bzw. initialisiert deaktivierte GUI-Elemente

Beide Flags können über ein logisches oder miteinander verknüpft werden.

## DefaultValueProvider

Wenn bei der Initialisierung eines DataItems ein Property gesetzt werden soll, die zu einem nicht kopierten GUI-Element gehört. Kann durch die Implementierung des Interface *IDefaulValueProvider* ein Objekt erzeugt werden, das einen Defaultwert zur Verfügung stellt. In diesem Fall muss lediglich die Methode Object GetDefaultValue() implementiert werden.

***Achtung: Typen, die das Interface IDefaultValueProvider implementieren müssen entweder einen öffentlichen, parameterlosen Konstrukt, oder die statische und öffentliche Methode GetInstance() aufweisen! Falls diese Methode existiert wird diese anstelle des Konstruktors aufgerufen.***

## Beispiel:

public class TestDataItem

{

private String value1;

private int value2;

private int value3;

public String Value1 { get { return value1; } set { value1 = value; } }

public int Value2 { get { return value2; } set { value2=value; } }

public int Value3 { get { return value3; } set { value3 = value; } }

}

partial class AInputForm

{

…

private System.Windows.Forms.Label label1;

private System.Windows.Forms.Label label2;

//Hier wird eine Verbindung zwischen der Property "Text" des Controls //textBox1 und der Property "Value1" eines Objekts vom Typ //TestDataItem hergestellt. Als Konverter wird die Implementierung //DummyConverter verwendet. Das Kopieren findet auch dann statt, wenn //das GUI-Element unsichtbar oder deaktiviert ist

[Conntection(typeof(TestDataItem),

"Text", "Value1", typeof(DummyConverter),

ConnectionFlag=ConnectionFlags.CopyOnDisabled |

ConnectionFlags.CopyOnInvisible)]

private System.Windows.Forms.TextBox textBox1;

//Im Grunde das Gleiche wie oben,jedoch wird hier die //Implementierung des IntStringConverters verwendet. Der //Kopiervorgang entfällt, falls das GUI-Element unsichtbar oder //deaktiviert ist.

[Conntection(typeof(TestDataItem),

"Text", "Value2",

typeof(IntStringConverter))]

private System.Windows.Forms.TextBox textBox2;

private System.Windows.Forms.Button button1;

private System.Windows.Forms.Button button2;

//Siehe oben, zusätzlich wurde noch ein DefaultValueProvider //spezifiziert

[Conntection(typeof(TestDataItem),

"Text", "Value3", typeof(IntStringConverter), DefaultValueProvider=typeof(DefaultProvider))]

private System.Windows.Forms.TextBox textBox3;

private System.Windows.Forms.Label label3;

…

}

class DefaultProvider : IDefaultValueProvider

{

//Da die Methode GetInstance() existiert wird der Konstruktor nicht

//verwendet

public static DefaultProvider GetInstance()

{

return new DefaultProvider();

}

public Object GetDefaultValue()

{

return 33;

}

}

//Wandelt Strings in Ints und umgekehrt um

class IntStringConverter : IValueConverter

{

public Object ConvertForDataItem(Object val)

{

String tmp = val.ToString();

return int.Parse(tmp);

}

public Object ConvertForControl(Object val)

{

return val.ToString();

}

}

//Nimmt keine Typkonvertierung vor

class DummyConverter : IValueConverter

{

public Object ConvertForDataItem(Object val)

{

return val;

}

public Object ConvertForControl(Object val)

{

return val;

}

}

Aus den Benutzereingabedaten kann nun mittels der DataItemFactory ein Objekt erzeugt werden:

DataItemFactory f = new DataItemFactory();

TestDataItem t = f.CreateDataItem<TestDataItem>(this);

Oder aus gegebenen Daten kann eine GUI initialisiert werden:

TestDataItem t = new TestDataItem();

t.Value1 = "Hallo";

t.Value2 = 22;

ControlConfigurator c = new ControlConfigurator();

c.ConfigureControl(this, t);

# Layoutmanager

Mithilfe der Factories und dem Kontexmanagement lassen sich Steuerelemente mittels Daten aus Objekten und Konfigurationdateien initialisieren und an- bzw. abschalten. Allerdings hinterlässt ein unsichtbar gemachtes Steuerelement in einer Form einen leeren Raum. Um alle Elemente einer Form neu anzuordnen enthält das *UMF* ein Layoutmanagementsystem. Die Anwendung dieses Layoutmanagers ist sehr einfach und lässt sich nahtlos in das Kontextmanagement einfügen.

## Anwendung des Layoutmangers

Damit das Layout eine Form vom Layoutmanager verwaltet werden kann sind zwei Schritte notwendig. Erstens ist die Auszeichnung der Form, dass sie dem Layoutmanagement unterworfen wird und mit welchem Manager dies geschehen soll, notwendig. Zweitens muss ein konkreter Layoutmanager erzeugt werden mit dessen Methoden das Verhalten der Form gesteuert werden kann, wie das folgende Beispiel zeigt:

[Layout(typeof(BoxLayoutFactory))]

public partial class MyForm : Form

{

private ILayout vc;

public MyForm()

{

InitializeComponent();

vc = LayoutFactory.Instance.CreateLayout(this);

…

}

…

}

Über das Attribut *Layout* wird dem dem Framework mitgeteilt, welchen Manager es verwenden soll. Momentan wird lediglich der ein BoxLayout unterstützt. Das Feld *vc* vom Typ *ILayout* dient dazu den erzeugten Layoutmanger zu referenzieren und zu verwenden. Schließlich wird über den Aufruf *CreateLayout* ein Layoutmanager für das entsprechende *Form* erzeugt.

Das Layoutmanagement kann jetzt über das von *vc* referenzierte Objekt gesteuert werden, dafür stellt das Interface *ILayout* mehrere Methoden zur Verfügung.

* void Pack()– Über diese Methode wird das Framework dazu veranlasst eine Form zu packen, das heißt Leerräume zu entfernen oder überlappende Elemente zu reorganiesieren. Dies ist wahrscheinlich die wichtigste Methode für den Anwender.
* Void Remove(Control c) – entfernt ein *Control* aus dem Layoutmanagementsystem. Dieses Element wird nicht mehr dem Layoutmanagement unterworfen. Sichtbare und verwendete Controls sollten nicht aus dem Layoutmanagement herausgenommen werden! Diese Methode dient dazu Elemente zu entfernen, die auch tatsächlich aus der *Form* entfernt wurden und der GarbageCollection zur Verfügung stehen sollen.
* int ComputedWidth { get; } - gibt die neue Breite einer Form zurück, falls *Pack* aufgerufen wird
* int ComputedWidth { get; } - gibt die neue Höhe einer Form zurück, falls *Pack* aufgerufen wird

## Integration ist Kontextmanagement

Über das Kontextmanagement können extern gesteuert *Controls* aktiviert und deaktiviert werden. Dies hat sehr oft zur Folge, dass eine Form neu gepackt werden muss, aus diesem Grund wurde der Kontextmanager so entworfen, dass der Layoutvorgang einer Form automatisiert werden kann. Dazu sind lediglich zwei weitere Schritte notwendig. Zum einen muss die Form das Interface *ILayouted* implementieren, zum anderen muss die Form beim Kontextmanager registriert werden.

[Layout(typeof(BoxLayoutFactory))]

public partial class MyForm : Form, **ILayouted**

{

private ILayout vc;

public MyForm()

{

InitializeComponent();

vc = LayoutFactory.Instance.CreateLayout(this);

**ContextManager.Instance.Register(this, vc);**

…

}

**public ILayout Layout { get { return vc; } }**

…

}

Das Interface *ILayouted* erzwingt die Implementierung eines Getters, der den Layoutmanger des Forms zurückliefert. Mittels der Methode *Register*, wird das Form für das automatische Layout im Kontextmanager registriert. Wird über den Kontextmanager der Kontext geändert, werden automatisch alle registrierten Form einem Layout unterworfen.

**Achtung:** Forms müssen Falls sie nicht mehr verwendet werden über die Methoden *UnregisterAll* des Kontextmanagers wieder freigegeben werden. Ansonsten stehen diese der Garbagecollection nicht zur Verfügung:

ContextManager.Instance.UnRegisterAll(…);

## Der Layoutvorgang

Der Layoutmanager von *UMF* ist so konzipiert, dass im VisualStudio zuerst ein Form entworfen werden kann und dann nachträglich aus der Anordnung der Steuerelemente des Forms das Layout angepasst wird. Das bedeutet, dass das Layoutsystem erst nach dem Entwurf automatisch aus dem gegebenen Form erzeugt wird.

Im Moment ist lediglich ein BoxLayoutManager implementiert, dessen Funktionsweise im Folgenden beschrieben wird. Der Vorgang läuft in der Erzeugungsphase des Layoutmanagers wie folgt ab:

1. Unterteile die Steuerelemente in vertikale Gruppen. Eine solche Gruppe enthält Steuerelemente, die sich in der vertikal überschneiden. Für alle nicht überschneidenden Elemente wird eine neue Gruppe gebildet.
2. Falls es vertikale Gruppen mit mehr als einem Steuerelement gibt, unterteile diese vertikalen Gruppen nach obigem Schema in horizontale Gruppen. Falls es horizontale Gruppen mit mehr als einem Element gibt, werden diese wie in 1. Zerlegt.

Dieser Vorgang wird rekursiv wiederholt, bis jede Gruppe nur noch ein *Control* enthält. Diese so entstandenen Gruppen werden als Boxen bezeichnet -> BoxLayoutManager.

Der pack()-Aufruf bewirkt, dass zuerst in die innersten Boxen abgestiegen wird. Dann wird überprüft, ob das das Steuerelement sichtbar ist und welche tatsächliche Größe es momentan hat. Danach wird die in äußere Gruppe gegangen, die nächste Untergruppe analysiert. Mit den angesammelten Daten wird dann die äußere Gruppe in ihrer Größe angepasst. Dieser Vorgang erfolgt rekursiv, bis der Algorithmus an den äußersten Gruppen angelangt ist. Es gilt zu beachten, dass vertikale Gruppen nur vertikal gepackt werden und horizontale Gruppen nur horizontal. Das heißt die Anordnung der Steuerelemente beeinflusst den Packvorgang.

## Steuerung des Layoutvorgangs über Panels

Ein Spezialfall stellt die Unterklasse *Panel* von *Control* dar. Ein *Panel* wird vom BoxLayoutManager nicht weiter zerlegt. Das bedeutet, dass der Analysealgorithmus bei einem *Panel* abbricht und dieses in eine Gruppe einfügt wenn sich nicht mehr mit anderen *Controls* außerhalb des *Panels* überschneidet. Dabei kann das Panel selbst wieder Steuerelement enthalten. Für den Layoutvorgang bedeutet das, dass Elemente, die in einem Panelverwaltet werden, nicht dem Layoutmanager des Forms unterworfen sind und das Panel, als eigenständiges *Control* betrachtet wird.

Es besteht aber die Möglichkeit ein Panel dem Layoutmanager hinzuzufügen, dazu wird es in gleicher Weise, wie die Form ausgezeichnet:

[Layout(typeof(BoxLayoutFactory))]

private System.Windows.Forms.Panel panel1;

Im Falle, dass für das enthaltende *Form* ein Layoutmanager erzeugt wird, passiert jetzt folgendes:

* Das *Panel* wird nach obigen Schema solange in Gruppen verschoben, bis es sich alleine in einer Gruppe befindet.
* Dann wird für das *Panel* intern ein neuer BoxLayoutManager erzeugt. Die Elemente in einem *Panel* werden, dann unabhängig davon ob sich das *Panel* in einer horizontalen oder vertikalen Gruppe befindet, nach obigem Algorithmus zuerst in horizontale Gruppen unterteilt.

Mittels Panels kann also das Verhalten des Layoutmanagers stark beeinflusst und auf Benutzerwünsche angepasst werden.

## Weitere Steuerungsmöglichkeiten

Über das Attribut *FixedLayoutGroup* kann das Verhalten des BoxLayoutmanagers weiter gesteuert werden. Mittels des Attributs können einzelne benachbarte Steuerelemente zu Gruppen zusammengefasst werden, die sich nicht in Form Größe ändern, solange sich nicht alle Elemente geändert haben. Dazu müssen die entsprechenden Elemente mit besagtem Attribut unter Angabe eines eindeutigen Namens ausgezeichnet werden.

Nehmen wir an, dass wir eine *Form* mit zwei nebeneinander liegenden *Buttons* *b1*, *b2* und weiteren Elementen definiert haben, der ein Layoutmanager zugeordnet wurde. Der Layoutvorgang soll aber falls *b1* unsichtbar ist *b2* nicht verschieben, so kann diese Ergebnis mit folgender Auszeichnung erreicht werden:

[FixedLayoutGroup("b1\_b2\_group")]

private System.Windows.Forms.Button b1;

[FixedLayoutGroup("b1\_b2\_group")]

private System.Windows.Forms.Button b2;

Beide *Buttons* befinden sich jetzt in der Gruppe *b1\_b2\_group* bei einer Größen- oder Sichtbarkeitänderung von einem der beiden Knöpfe, wird der andere nicht verschoben.

Falls noch mehr Steuerungsmöglichkeiten für das Layouting vonnöten sind, gibt es nich die Möglichkeit einer *FixedLayoutGroup* ein Steuerobjekt vom Typ *ICustomLayoutControl* hinzuzufügen. Implementierung dieses Interfaces können benutzerdefinierten Code enthalten, der während des Layoutvorgangs ausgeführt wird. Alle Implementierung des Interfaces müssen folgende Methoden beachten:

* Control DeclaringControl { get; set; } – Setzt das deklarierende *Control*  der aktuellen Gruppe. Falls sich also ein *Button*  in einer Gruppe befindet, der in einem *Form* deklariert wurde. So das *Form* das deklarierende Element.
* void PrePack() – Hier kann Code eingefügt der vor dem Packen der Gruppe ausgeführt wird
* void PostPack() – Hier kann Code eingefügt der nach dem Packen der Gruppe ausgeführt wird
* bool DoLayout() – Gibt zurück, ob der Layoutvorgang für die Gruppe ausgeführt werden soll oder nicht.

Die Auszeichnung geschieht dann wie im folgeden Beispiel:

private class Custom : ICustomLayoutControl

{

…

}

…

[FixedLayoutGroup("b1\_b2\_group", CustomLayoutControl=typeof(Custom))]

private System.Windows.Forms.Button b1;

[FixedLayoutGroup("b1\_b2\_group")]

private System.Windows.Forms.Button b2;

Der Code der in *Custom* definiert wurde, wird ab jetzt immer dann ausgeführt, wenn der Layoutmanager die Gruppe *b1\_b2\_group* packt.

## Pitfalls

Momentan ist nur eine Pitfall bekannt. Der Analysealgorithmus des BoxLayoutManagers bricht nicht ab, sobald sich Steuerelemente überschneiden und führt zu einem Stackoverflow.

Für weitere Fragen bitte an [stork.daniel@googlemail.com](mailto:stork.daniel@googlemail.com) schreiben.